GALAHAD ROOTS

USER DOCUMENTATION GALAHAD Optimization Library version 5.1

1 SUMMARY

This package uses classical formulae together with Newton's method to find all the real roots of a real polynomial.

ATTRIBUTES — Versions: `GALAHAD_ROOTS_single, GALAHAD_ROOTS_double`. **Uses:** `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_SORT`. **Date:** November 2010. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the `USE` statement

```
USE GALAHAD_ROOTS_single
```

with the obvious substitution `GALAHAD_ROOTS_double`, `GALAHAD_ROOTS_quadruple`, `GALAHAD_ROOTS_single_64`, `GALAHAD_ROOTS_double_64` and `GALAHAD_ROOTS_quadruple_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived types `ROOTS_control_type`, `ROOTS_inform_type` and `ROOTS_data_type` (Section 2.2) and the subroutine `ROOTS_solve`, (Section 2.3) must be renamed on one of the `USE` statements.

2.1 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords `REAL(rp_)` and `INTEGER(ip_)`, where `rp_` and `ip_` are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default `REAL` for the single precision versions, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined by the fortran `iso_fortran_env` module. The latter are default (32-bit) and long (64-bit) integers, and correspond to `ip_ = int32` and `ip_ = int64`, respectively, again from the `iso_fortran_env` module.

2.2 The derived data types

Three derived data types are accessible from the package.

2.2.1 The derived data type for holding control parameters

The derived data type `ROOTS_control_type` is used to hold controlling data. Default values may be obtained by calling `ROOTS_initialize` (see Section 2.3.1), while components may also be changed by calling `ROOTS_read_specfile` (see Section 2.5.1). The components of `ROOTS_control_type` are:

`error` is a scalar variable of type `INTEGER(ip_)`, that holds the stream number for error messages. Printing of error messages in `ROOTS_solve` and `ROOTS_terminate` is suppressed if `error` ≤ 0. The default is `error = 6`.

`out` is a scalar variable of type `INTEGER(ip_)`, that holds the stream number for informational messages. Printing of informational messages in `ROOTS_solve` is suppressed if `out` < 0. The default is `out = 6`.

print_level is a scalar variable of type INTEGER(ip_), that is used to control the amount of informational output which is required. No informational output will occur if print_level ≤ 0. If print_level ≥ 1, debugging information will be provided. The default is print_level = 0.

tol is an INTENT(IN) scalar of type REAL(rp_) that should be set to the required accuracy of the roots. Every effort will be taken to ensure that each computed root x_c lies within $\pm$ tol x_e of its exact equivalent x_e, although sometimes the required accuracy will not be possible. The default is tol = EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_ROOTS_double).

space_critical is a scalar variable of type default LOGICAL, that must be set .TRUE. if space is critical when allocating arrays and .FALSE. otherwise. The package may run faster if space_critical is .FALSE. but at the possible expense of a larger storage requirement. The default is space_critical = .FALSE..

deallocate_error_fatal is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to terminate execution if a deallocation fails, and .FALSE. if an attempt to continue will be made. The default is deallocate_error_fatal = .FALSE..

prefix is a scalar variable of type default CHARACTER and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string prefix(2:LEN(TRIM(prefix))-1), thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default prefix = "".

2.2.2 The derived data type for holding informational parameters

The derived data type ROOTS_inform_type is used to hold parameters that give information about the progress and needs of the algorithm. The components of ROOTS_inform_type are:

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See Section 2.4 for details.

alloc_status is a scalar variable of type INTEGER(ip_), that gives the status of the last attempted array allocation or deallocation. This will be 0 if status = 0.

bad_alloc is a scalar variable of type default CHARACTER and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if status = 0.

2.2.3 The derived data type for holding problem data

The derived data type ROOTS_data_type is used to hold all the data for a particular problem, between calls of ROOTS procedures. This data should be preserved, untouched, from the initial call to ROOTS_initialize to the final call to ROOTS_terminate.

2.3 Argument lists and calling sequences

There are three procedures for user calls (see Section 2.5 for further features):

1. The subroutine ROOTS_initialize is used to set default values, and initialize private data.
2. The subroutine ROOTS_solve is called to find the real roots of the polynomial

$$\sum_{i=0}^{d} a_i x^i \tag{2.1}$$

   of degree d, where the coefficients a_i, $0 \leq i \leq d$ are real.

3. The subroutine ROOTS_terminate allows the user to automatically deallocate array components of the private data, allocated by ROOTS_solve, at the end of the solution process.

2.3.1 The initialization subroutine

Default values are provided as follows:

```
CALL ROOTS_initialize( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type ROOTS_data_type (see Section 2.2.3). It is used to hold data about the problem being solved.

control is a scalar INTENT(OUT) argument of type ROOTS_control_type (see Section 2.2.1). On exit, control contains default values for the components as described in Section 2.2.1. These values should only be changed after calling ROOTS_initialize.

inform is a scalar INTENT(OUT) argument of type ROOTS_inform_type (see Section 2.2.2). A successful call to ROOTS_initialize is indicated when the component status has the value 0.

2.3.2 The solution subroutine

The roots of the polynomial (2.1) are found as follows

```
CALL ROOTS_solve( A, nroots, ROOTS, control, inform, data )
```

A is an INTENT(IN) rank-one array of type REAL(rp_), whose lower bound must be 0 and whose upper bound specifies the degree, d, of the polynomial. The entries A$(i), i = 0, \ldots,$ UBOUND(A), must contain the values of the real coefficients a_i, $0 \leq i \leq d$. **Restrictions:** UBOUND(A,1) ≥ 0.

nroots is an INTENT(OUT) scalar of type INTEGER(ip_), that gives the number of real roots of the polynomial.

ROOTS is an INTENT(OUT) rank-one array of length d and type REAL(rp_). On exit, ROOTS(:nroots) give the values of the real roots of the polynomial in increasing order. **Restrictions:** SIZE(ROOTS) $\geq$ UBOUND(A,1).

control is a scalar INTENT(IN) argument of type ROOTS_control_type (see Section 2.2.1). Default values may be assigned by calling ROOTS_initialize prior to the first call to ROOTS_solve.

inform is a scalar INTENT(INOUT) argument of type ROOTS_inform_type (see Section 2.2.2). A successful call to ROOTS_solve is indicated when the component status has the value 0. For other return values of status, see Section 2.4.

data is a scalar INTENT(INOUT) argument of type ROOTS_data_type (see Section 2.2.3). It is used to hold data about the problem being solved. It must not have been altered **by the user** since the last call to ROOTS_initialize.

2.3.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

```
CALL ROOTS_terminate( data, control, inform )
```

data is a scalar INTENT(INOUT) argument of type ROOTS_data_type exactly as for ROOTS_solve, which must not have been altered **by the user** since the last call to ROOTS_initialize. On exit, array components will have been deallocated.

control is a scalar INTENT(IN) argument of type ROOTS_control_type exactly as for ROOTS_solve.

inform is a scalar INTENT(OUT) argument of type ROOTS_inform_type exactly as for ROOTS_solve. Only the component status will be set on exit, and a successful call to ROOTS_terminate is indicated when this component status has the value 0. For other return values of status, see Section 2.4.

2.4 Warning and error messages

A negative value of `inform%status` on exit from `ROOTS_solve` indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

`-1.` An allocation error occured. A message indicating the offending array is written on unit `control%error`, and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_-status` and `inform%bad_alloc` respectively.

`-2.` A deallocation error occured. A message indicating the offending array is written on unit `control%error` and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_-status` and `inform%bad_alloc` respectively.

`-3.` Either the specified degree of the polynomial in `degree` is less than 0, or the declared dimension of the array `ROOTS` is smaller than the specified degree.

2.5 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `ROOTS_control_type` (see Section 2.2.1), by reading an appropriate data specification file using the subroutine `ROOTS_read_specfile`. This facility is useful as it allows a user to change `ROOTS` control parameters without editing and recompiling programs that call `ROOTS`.

A specification file, or specfile, is a data file containing a number of ”specification commands”. Each command occurs on a separate line, and comprises a ”keyword”, which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) ”value”, which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by `ROOTS_read_specfile` must start with a ”`BEGIN ROOTS`” command and end with an ”`END`” command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by CQP_read_specfile .. )
  BEGIN CQP
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by CQP_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The “`BEGIN ROOTS`” and “`END`” delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

```
  BEGIN CQP SPECIFICATION
```

and

```
  END CQP SPECIFICATION
```

are acceptable. Furthermore, between the “`BEGIN ROOTS`” and “`END`” delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is `!` or `*` are ignored. The content of a line after a `!`

or `*` character is also ignored (as is the `!` or `*` character itself). This provides an easy manner to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are "`ON`", "`TRUE`", "`.TRUE.`", "`T`", "`YES`", "`Y`", or "`OFF`", "`NO`", "`N`", "`FALSE`", "`.FALSE.`" and "`F`". Empty values are also allowed for logical control parameters, and are interpreted as "`TRUE`".

The specification file must be open for input when `ROOTS_read_specfile` is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is `REWIND`ed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by `ROOTS_read_specfile`.

2.5.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL CQP_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `ROOTS_control_type` (see Section 2.2.1). Default values should have already been set, perhaps by calling `ROOTS_initialize`. On exit, individual components of `control` may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.2.1) of `control` that each affects are given in Table 2.1.

command	component of `control`	value type
`error-printout-device`	`%error`	integer
`printout-device`	`%out`	integer
`print-level`	`%print_level`	integer
`root-tolerance`	`%tol`	real
`space-critical`	`%space_critical`	logical
`deallocate-error-fatal`	`%deallocate_error_fatal`	logical
`output-line-prefix`	`%prefix`	`character`

Table 2.1: Specfile commands and associated components of `control`.

`device` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set to the unit number on which the specfile has been opened. If `device` is not open, `control` will not be altered and execution will continue, but an error message will be printed on unit `control%error`.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `ROOTS_solve` calls the GALAHAD packages `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SPECFIL`
and `GALAHAD_SORT`.

Input/output: Output is under control of the arguments `control%error`, `control%out` and `control%print_level`.

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 METHOD

Littlewood and Ferrari's algorithms are used to find estimates of the real roots of cubic and quartic polynomials, respectively; a stabilized version of the well-known formula is used in the quadratic case. Newton's method and/or methods based on the companion matrix are used to further refine the computed roots if necessary. Madsen and Reid's (1975) method is used for polynomials whose degree exceeds four.

References:

The basic method is that given by

K. Madsen and J. K. Reid, "FORTRAN Subroutines for Finding Polynomial Zeros". Technical Report A.E.R.E. R.7986, Computer Science and System Division, A.E.R.E. Harwell, Oxfordshire, U.K. (1975)

5 EXAMPLE OF USE

Suppose we wish to solve the quadratic, cubic, quartic and quintic equations

$$\begin{aligned} x^2 - 3x + 2 &= 0 \\ x^3 - 6x^2 + 11x - 6 &= 0 \\ x^4 - 10x^3 + 35x^2 - 50x + 24 &= 0 \text{ and} \\ x^5 - 15x^4 + 85x^3 - 225x^2 + 274x - 120 &= 0. \end{aligned}$$

Then we may use the following code:

```
! THIS VERSION: GALAHAD 2.1 - 22/03/2007 AT 09:00 GMT.
   PROGRAM GALAHAD_ROOTS_EXAMPLE
   USE GALAHAD_ROOTS_double         ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )  ! set precision
   REAL ( KIND = wp ), PARAMETER :: one = 1.0_wp
   INTEGER :: degree, nroots
   REAL ( KIND = wp ) :: A( 0 : 5 ), ROOTS( 5 )
   TYPE ( ROOTS_data_type ) :: data
   TYPE ( ROOTS_control_type ) :: control
   TYPE ( ROOTS_inform_type ) :: inform
   control%tol = EPSILON( one ) ** 0.75       ! accuracy requested
   DO degree = 2, 5                           ! polynomials of degree 2 to 5
     IF ( degree == 2 ) THEN
       A( 0 ) = 2.0_wp
       A( 1 ) = - 3.0_wp
       A( 2 ) = 1.0_wp
       WRITE( 6, "( ' Quadratic ' )" )
       CALL ROOTS_solve( A( : degree ), nroots, ROOTS( : degree ),             &
                         control, inform, data )
     ELSE IF ( degree == 3 ) THEN
       A( 0 ) = - 6.0_wp
       A( 1 ) = 11.0_wp
       A( 2 ) = - 6.0_wp
       A( 3 ) = 1.0_wp
       WRITE( 6, "( /, ' Cubic ' )" )
       CALL ROOTS_solve( A( : degree ), nroots, ROOTS( : degree ),             &
                         control, inform, data )
     ELSE IF ( degree == 4 ) THEN
```

```
         A( 0 ) = 24.0_wp
         A( 1 ) = - 50.0_wp
         A( 2 ) = 35.0_wp
         A( 3 ) = - 10.0_wp
         A( 4 ) = 1.0_wp
         WRITE( 6, "( /, ' Quartic ' )" )
         CALL ROOTS_solve( A( : degree ), nroots, ROOTS( : degree ),             &
                           control, inform, data )
       ELSE IF ( degree == 5 ) THEN
         A( 0 ) = - 120.0_wp
         A( 1 ) = 274.0_wp
         A( 2 ) = - 225.0_wp
         A( 3 ) = 85.0_wp
         A( 4 ) = - 15.0_wp
         A( 5 ) = 1.0_wp
         WRITE( 6, "( /, ' Quintic ' )" )
         CALL ROOTS_solve( A( : degree ), nroots, ROOTS( : degree ),             &
                           control, inform, data )
       END IF
       IF ( nroots == 0 ) THEN
         WRITE( 6, "( ' no real roots ' )" )
       ELSE IF ( nroots == 1 ) THEN
         WRITE( 6, "( ' 1 real root ' )" )
       ELSE IF ( nroots == 2 ) THEN
         WRITE( 6, "( ' 2 real roots ' )" )
       ELSE IF ( nroots == 3 ) THEN
         WRITE( 6, "( ' 3 real roots ' )" )
       ELSE IF ( nroots == 4 ) THEN
         WRITE( 6, "( ' 4 real roots ' )" )
       ELSE IF ( nroots == 5 ) THEN
         WRITE( 6, "( ' 5 real roots ' )" )
       END IF
       IF ( nroots /= 0 ) WRITE( 6, "( ' roots: ', 5ES10.2 )" ) ROOTS( : nroots )
     END DO
     END PROGRAM GALAHAD_ROOTS_EXAMPLE
```

This produces the following output:

```
 Quadratic
 2 real roots
 roots:   1.00E+00  2.00E+00

 Cubic
 3 real roots
 roots:   1.00E+00  2.00E+00  3.00E+00

 Quartic
 4 real roots
 roots:   1.00E+00  2.00E+00  3.00E+00  4.00E+00

 Quintic
 5 real roots
 roots:   1.00E+00  2.00E+00  3.00E+00  4.00E+00  5.00E+00
```